



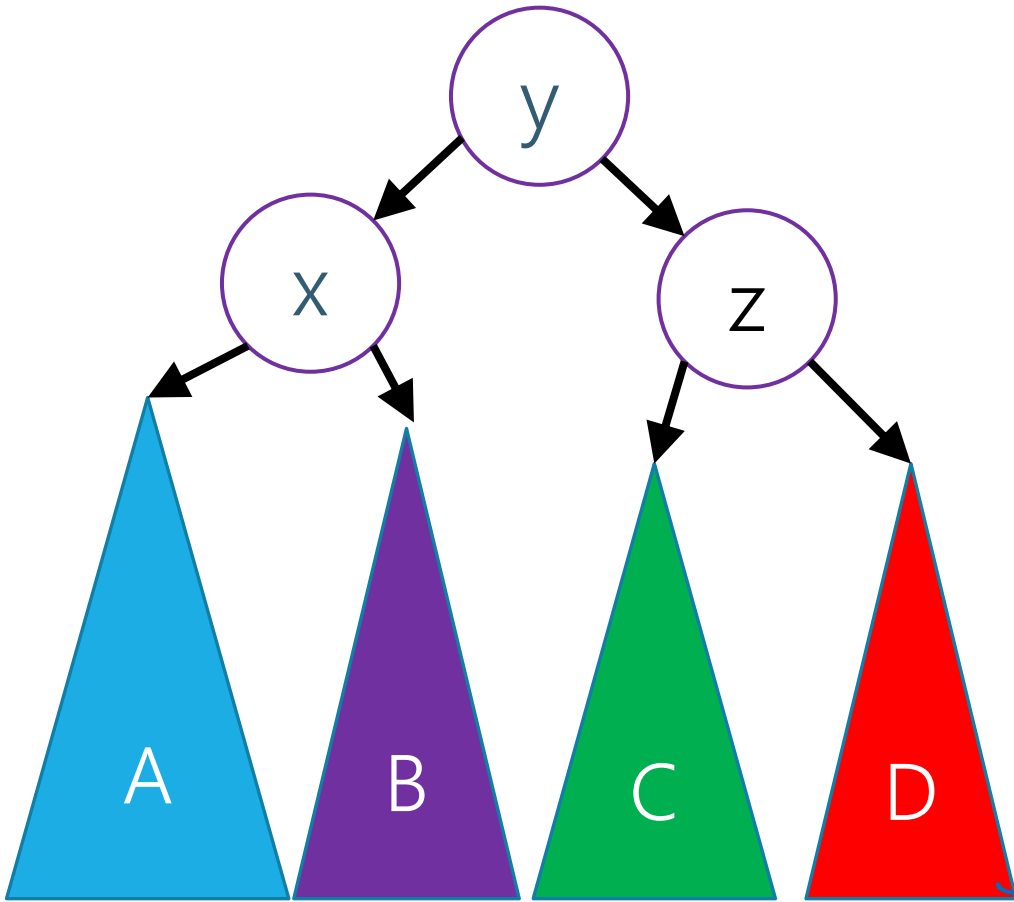
# Wrap AVL, Hashing

CSE 332 Spring 25  
Lecture 9

# Logistics

	Monday	Tuesday	Wednesday	Thursday	Friday
This Week	Ex 2 due				<b>TODAY</b> Ex 3 due Ex 4 out
Next Week	Ex 5 out				Ex 4 due

# Four Types of Rotations



Insert location (relative to lowest imbalanced node)	Solution
Left subtree of left child (A)	Single right rotation
Right subtree of left child (B)	Double (left-right) rotation
Left subtree of right child (C)	Double (right-left) rotation
Right subtree of right child(D)	Single left rotation

# How Long Does Rebalancing Take?

Assume we store in each node the height of its subtree.

How do we find an unbalanced node?

How many rotations might we have to do?

# How Long Does Rebalancing Take?

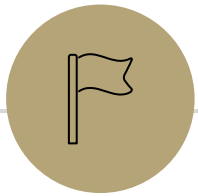
Assume we store in each node the height of its subtree.

How do we find an unbalanced node?

- Just go back up the tree from where we inserted.

How many rotations might we have to do?

- Just a single or double rotation on the lowest unbalanced node.
- A rotation will cause the subtree rooted where the rotation happens to have the same height it had before insertion.



## Some Related Topics

Lazy deletion, formally arguing that we have height  $\log n$

# Lazy Deletion

Lazy Deletion: A general way to make `delete()` more efficient.  
(specifically, as efficient as `find()`)

Don't remove the entry from the structure, just "mark" it as deleted.

## Benefits:

→ Much simpler to implement

→ More efficient to delete (no need to shift values on every single delete)

Every node/array-index/etc. gets a Boolean.

## Drawbacks:

→ Extra space:

- For the flag

→ More drastically, data structure grows with all insertions, not with the current number of items.

→ Sometimes makes other operations more complicated (we'll see with hash tables).

# Simple Dictionary Implementations

	Insert	Find	Delete
Unsorted Linked List	$\Theta(m)$	$\Theta(m)$	$\Theta(m)$
Unsorted Array	$\Theta(m)$	$\Theta(m)$	$\Theta(m)$
Sorted Linked List	$\Theta(m)$	$\Theta(m)$	$\Theta(m)$
<del>Sorted Array</del>	$\Theta(m)$	$\Theta(\log m)$	$\Theta(\log m)$

We can do slightly better with **lazy deletion**, let  $m$  be the total number of elements ever inserted (even if later lazily deleted)  
Think about what happens if a **repeat key** is inserted!



# Deletion

In Exercise: Just do lazy deletion!

Alternatively: a similar set of rotations is possible to rebalance after a deletion.

- The textbook (or Wikipedia) can tell you more.
- The delete rotations are more involved—you may have to rotate multiple times up the tree. But you'll still do  $\Theta(\log n)$  rotations in total



**Formally bounding height**

# Where Were We?

We used rotations to restore the AVL property after insertion.

If  $h$  is the height of an AVL tree:

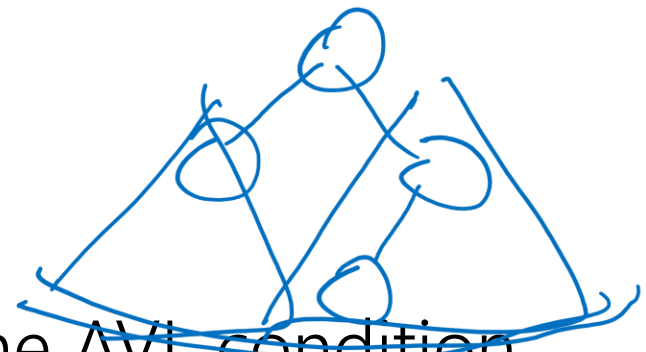
It takes  $O(h)$  time to find an imbalance (if any) and fix it.

So the worst case running time of insert?  $\Theta(h)$ .

Is  $h$  always  $O(\log n)$ ? YES! These are all  $\Theta(\log n)$ . Let's prove it!



# Bounding the Height $h=2?$



Suppose you have a tree of height  $h$ , meeting the AVL condition.

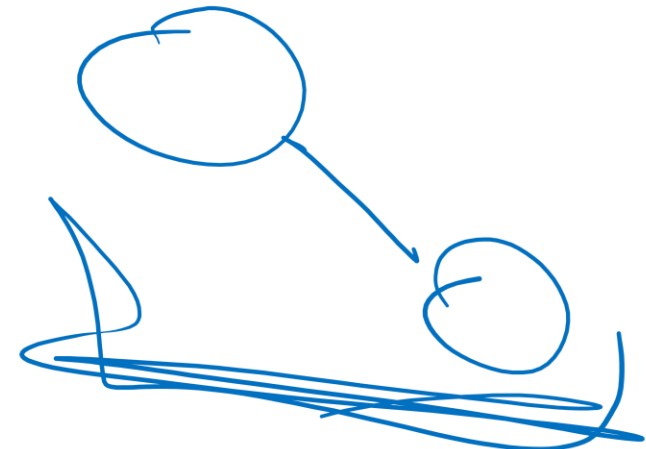
**AVL condition:** For every node, the height of its left subtree and right subtree differ by at most 1.

What is the minimum number of nodes in the tree?

If  $h = \underline{0}$ , then 1 node



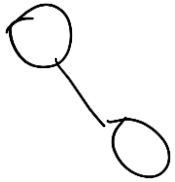
If  $\underline{h} = 1$ , then 2 nodes.



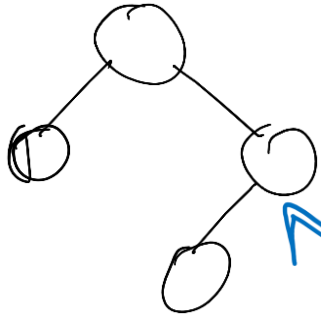
In general?

# Some Doodles

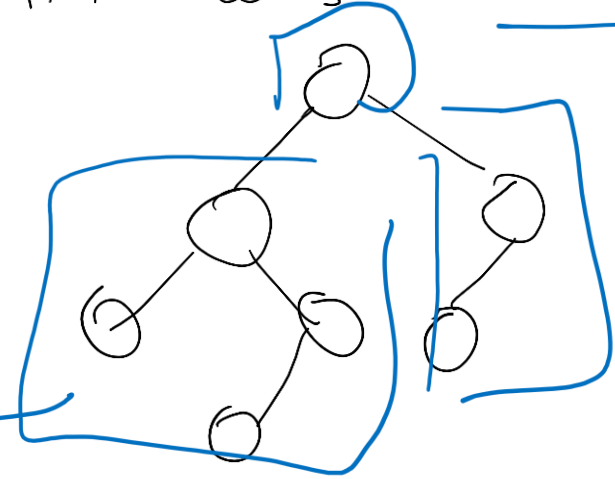
min nodes,  $h=1$



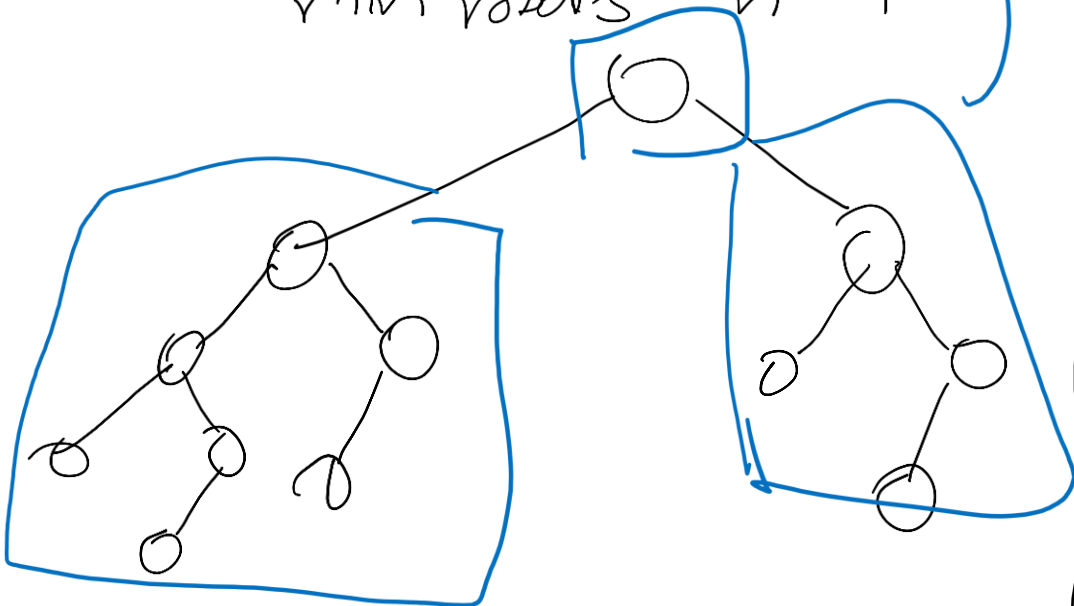
min nodes,  $h=2$



min nodes,  $h=3$



min nodes,  $h=4$

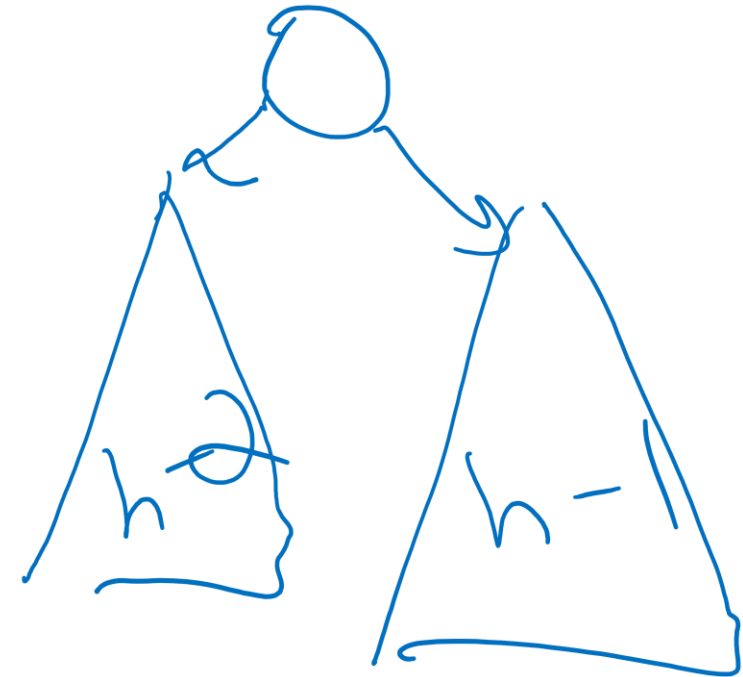


Might be sparser than you would have guessed!  
But it will still be logarithmic height...let's prove it!

# Bounding the Height

In general, let  $N()$  be the minimum number of nodes in a tree of height  $h$ , meeting the AVL requirement.

$$N(h) = \begin{cases} 1 & \text{if } h = 0 \\ 2 & \text{if } h = 1 \\ \underline{N(h-1) + N(h-2) + 1} & \text{otherwise} \end{cases}$$



# Bounding the Height

$$N(h) = \begin{cases} 1 & \text{if } h = 0 \\ 2 & \text{if } h = 1 \\ N(h-1) + N(h-2) + 1 & \text{otherwise} \end{cases}$$

We can try a recursion tree...

# Bounding the Height

$$N(h) = \begin{cases} 1 & \text{if } h = 0 \\ 2 & \text{if } h = 1 \\ N(h-1) + N(h-2) + 1 & \text{otherwise} \end{cases}$$

When evaluating, we'll quickly realize:

- Something with Fibonacci numbers is going on.
- It's going to be hard to exactly describe the pattern.

The real solution (using deep math magic beyond this course) is

$$N(h) \geq \phi^h - 1 \text{ where } \phi \text{ is } \frac{1+\sqrt{5}}{2} \approx 1.62$$

*golden ratio*



# The Proof

To convince you that the recurrence solution is correct, I don't need to tell you where it came from.

I just need to prove it correct via induction.

We'll need this fact:  $\phi + 1 = \phi^2$

It's easy to check by just evaluating  $\left(\frac{1+\sqrt{5}}{2}\right)^2$

A handwritten diagram in blue ink. It shows the equation  $x^2 = x + 1$ . Below the equation is a horizontal line. An arrow points from the right side of the equation ( $x + 1$ ) down to the horizontal line.

# The Proof

$$N(h) = \begin{cases} 1 & \text{if } h = 0 \\ 2 & \text{if } h = 1 \\ N(h-1) + N(h-2) + 1 & \text{otherwise} \end{cases}$$

$$\phi + 1 = \phi^2$$

Base Cases:  $\phi^0 - 1 = 0 < 1 = N(0)$     $\phi^1 - 1 = \phi - 1 \approx 0.62 < 2 = N(1)$

# Inductive Step

Inductive Hypothesis: Suppose that  $N(h) > \phi^h - 1$  for  $h < k$ .

Inductive Step: We show  $N(k) > \phi^k - 1$ .

$$N(k) = N(k-1) + N(k-2) + 1$$

$$> \phi^{k-1} - 1 + \phi^{k-2} - 1 + 1$$

$$= \phi^{k-1} + \phi^{k-2} - 1$$

$$= \phi^{k-2}(\phi + 1) - 1$$

$$= \phi^{k-2}(\phi^2) - 1$$

$$= \phi^{k+1} - 1$$

definition of  $N()$

by IH (note we need a strong hypothesis here)

algebra

fact from last slide

# What's the point?

$$\underline{N(h)} \leq \underline{\phi^h - 1}$$

The number of nodes in an AVL tree of height  $h$  is always at least  $\phi^h - 1$

So in an AVL tree with  $n$  elements, the height is always at most  $\log_{\phi}(n + 1)$

In big- $O$  terms, that's enough to say the height is  $\Theta(\log n)$ .

So our AVL trees really do have  $\Theta(\log n)$  worst cases for insert, find, and delete!



**Is the improvement worth it?**

---

# Wrap Up

$$\underline{\Theta(1)}$$

$$\underline{\Theta(\log n)}$$


$$\underline{\Theta(n)}$$

Was this...worth it?

That was a lot of work (and there are going to be a lot of pointers to move around to actually make those trees...)

Is going from  $n$  to  $\log n$  a good enough improvement?

# Wrap Up



	Insert	Find	Delete
Unsorted Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Unsorted Array	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Sorted Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Sorted Array	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$
AVL Tree	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

# Does It Matter?

$\Theta(\log n)$  is the most common running time between  $\Theta(1)$  and  $\Theta(n)$ .

Which of these times is  $\log n$  between?

$\Theta(1)$	A	$\Theta(n^{0.01})$	B	$\Theta(n^{0.1})$	C	$\Theta(n^{0.5})$	D	$\Theta(n)$
-------------	---	--------------------	---	-------------------	---	-------------------	---	-------------

Handwritten annotations:

- A bracket under  $\Theta(1)$  and  $\Theta(n)$  with arrows pointing to A and D.
- An arrow from A to  $\Theta(n^{0.01})$ .
- An arrow from  $\Theta(n^{0.01})$  to B.
- An arrow from B to  $\Theta(n^{0.1})$ .
- An arrow from  $\Theta(n^{0.1})$  to C.
- An arrow from C to  $\Theta(n^{0.5})$ .
- An arrow from  $\Theta(n^{0.5})$  to D.
- Handwritten  $\log_2(n)$  below the first three columns.
- Handwritten  $2^{\square} = n$  below the last three columns.



# Does It Matter?

We sped up from  $\Theta(n)$  to  $\Theta(\log n)$ .

Suppose we could handle an input of size  $k$  with the  $\Theta(n)$  algorithm. If the constant factors are small (and similar) we'll be able to handle an input of size  $2^k$  in the same time.

Imagine you can handle an input of size  $s$ . Can you handle an input of size  $s^2$ ?

For  $\Theta(n)$  algorithms, maybe not; it will almost square the time it takes.

- "Almost" is because the constant factors don't change.

For  $\Theta(\log n)$  algorithms, yes! You'll at most double the time it takes.

- "At most" because the lower order terms change differently.



# Does It Matter?

Let's say you have an algorithm that takes  $n$  milliseconds or  $\log_2 n$  milliseconds. How long does it take to run when  $n$  is...

	1,000	10,000	100,000	1,000,000	10,000,000	100,000,000
$\log_2(n)$	0.010 s	0.013s	0.017s	0.020s	0.023s	0.027s
$n$	1 second	10 seconds	2 minutes	17 minutes	2.7 hours	1 day

$\log_2(\text{number of particles in the universe}) \approx 268.$

Constant factors do matter, (and for reasons you'll learn in 351, constant factors get worse as your dataset gets bigger) but you can't make a dataset so big that the  $\log_2(n)$  part will be what makes it intractable.

# Other Dictionaries

There are lots of flavors of self-balancing search trees

“Red-black trees” work on a similar principle to AVL trees, similar tradeoffs.

“Splay trees”

- Get  $O(\log n)$  amortized bounds for all operations.

“Scapegoat trees”

“Treaps” – a BST and heap in one (!)

Similar tradeoffs to AVL trees.

Next: A completely different idea for a dictionary

Goal:  $O(1)$  operations on average, in exchange for  $O(n)$  worst case.

# Wrap Up

AVL Trees:

$O(\log n)$  worst case find, insert, and delete.

Pros:

- Much more reliable running times than regular BSTs.

Cons:

- Tricky to implement
- A little more space to store subtree heights



# A Different Dictionary

---

# Another Dictionary

Our guiding principle for designing AVL trees was optimizing for the worst case.

What if we want to optimize for the **average case**?

That goal will lead us to a totally different data structure: **hash tables**

# A Simple Case

Suppose you were promised your keys would be distinct numbers in the range 0 to  $k$ .

How would you implement a dictionary. What are the running times for insert, find, and delete?

Just store the values in an array of size  $k + 1$ .

Store the value associated with  $i$  at index  $i$  of the array.

$O(1)$  operations for everything!

$O(1)$

# Generalization (Step 1)

What if the keys are guaranteed to be integers,  
But the upper limit is huge.

Why not just use the array from last time?

How could we still use the array of size  $k$ ?



# Generalization (Step 1)

What if the keys are guaranteed to be integers,  
But the upper limit is huge.

Why not just use the array from last time?  
- WAY too much space

How could we still use the array of size  $k$ ?  
- Map the keys into the range  $\{0, \dots, k - 1\}$ .

# % table size

Map to index  $\text{key} \% \text{TableSize}$


indices	0	1	2	3	4	5	6	7	8	9
array	20, ":"	11, "biz"				5, "bar"			18, "bop"	

`put(0, "foo");`     $0 \% 10 = 0$

`put(5, "bar");`     $5 \% 10 = 5$

`put(11, "biz");`     $11 \% 10 = 1$

`put(18, "bop");`     $18 \% 10 = 8$

`put(20, ":" );`     $20 \% 10 = 0$   Collision!

Problem 1: What do we do when the keys collide?

# Collision Resolution

Multiple Possible Strategies.

We'll talk about "open addressing" strategies later.

First, we'll discuss "Separate Chaining"

Idea: If more than one thing goes to the same spot

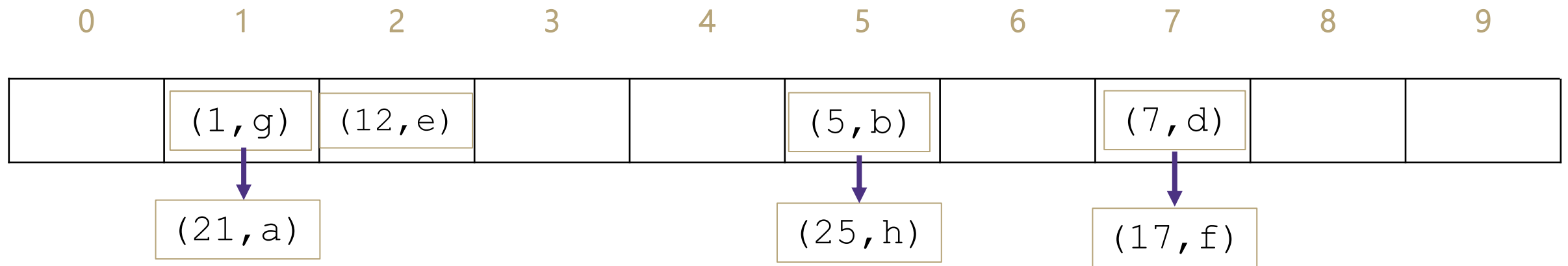
Just stuff them all in that one spot!

# Separate Chaining

Instead of an array of values

Have an array of (say) LinkedLists of values.

Insert the following keys: (1, a) (5,b) (21,a) (7,d) (12,e) (17,f) (1,g) (25,h)



# Running Times

What are the running times for:

`insert`

Best:

Worst:

`find`

Best:

Worst:

`delete`

Best:

Worst:

# Running Times

What are the running times for:

`insert`

Best:  $O(1)$

Worst:  $O(n)$

`find`

Best:  $O(1)$

Worst:  $O(n)$

`delete`

Best:  $O(1)$

Worst:  $O(n)$

# Average Case

What about on average?

Let's **assume** that the keys are randomly distributed

What is the average running time if the size of the table *TableSize* and we've inserted  $n$  keys?

insert

find

delete

# Average Case

What about on average?

Let's **assume** that the keys are (independently and uniformly) randomly distributed.

What is the average running time if the size of the table *TableSize* and we've inserted  $n$  keys?

insert  $O(1)$

find  $O\left(1 + \frac{n}{TableSize}\right)$

delete  $O\left(1 + \frac{n}{TableSize}\right)$



# Average Case

What about on average?

Let's **assume** that the keys are (independently and uniformly) randomly distributed.

What is the average running time if the size of the table *TableSize* and we've inserted  $n$  keys?

insert  $O(1)$

find  $O(1 + \lambda)$

delete  $O(1 + \lambda)$

We'll denote  $\frac{n}{TableSize}$  by  $\lambda$ .  
Often called "load factor"

# When $\lambda$ Grows

If we keep inserting things into the array,  $\lambda$  will keep increasing.

We'll never *really* run out of room.

When should we resize?

When it slows us down, i.e. when  $\lambda$  is a constant.

Heuristic: for separate chaining  $\lambda$  between 1 and 3 is a good time to resize.

# Resizing

How long does it take to resize?

Need to:

Remake the table

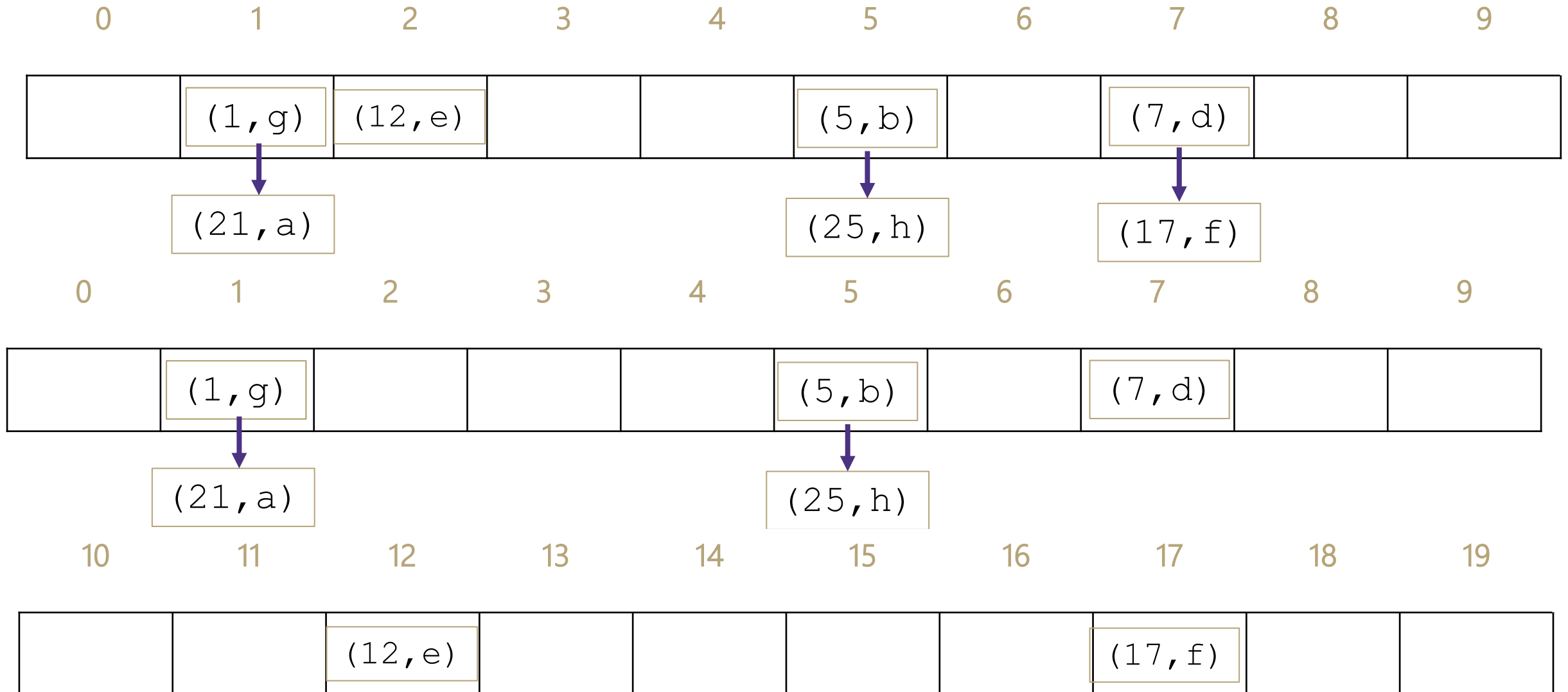
Evaluate the hash function over again.

Re-insert.

Total time:  $O(n + TableSize) = O(n)$  if  $\lambda$  is a constant.

# Resizing Redux

Let's resize by doubling the size of the array.



# Resizing Redux

That didn't work very well!

It turned out that most of the keys that were equal mod 10 were also equal mod 20.

This is likely with real data.

Don't just double the table size

Instead make the table size some new prime number.

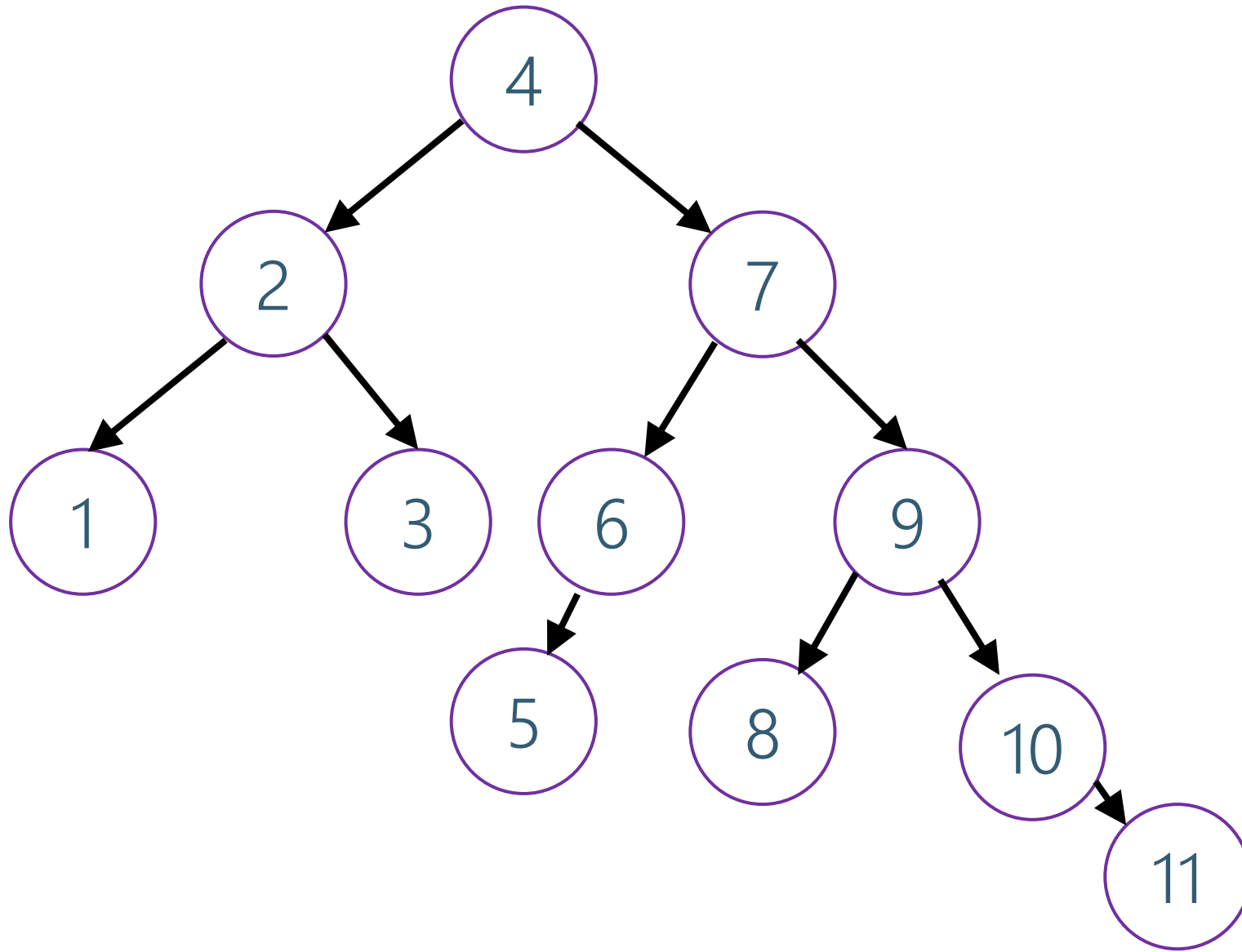
Collisions can still happen, but patterns with multiple prime numbers are rarer in real data than patterns with powers of 2.



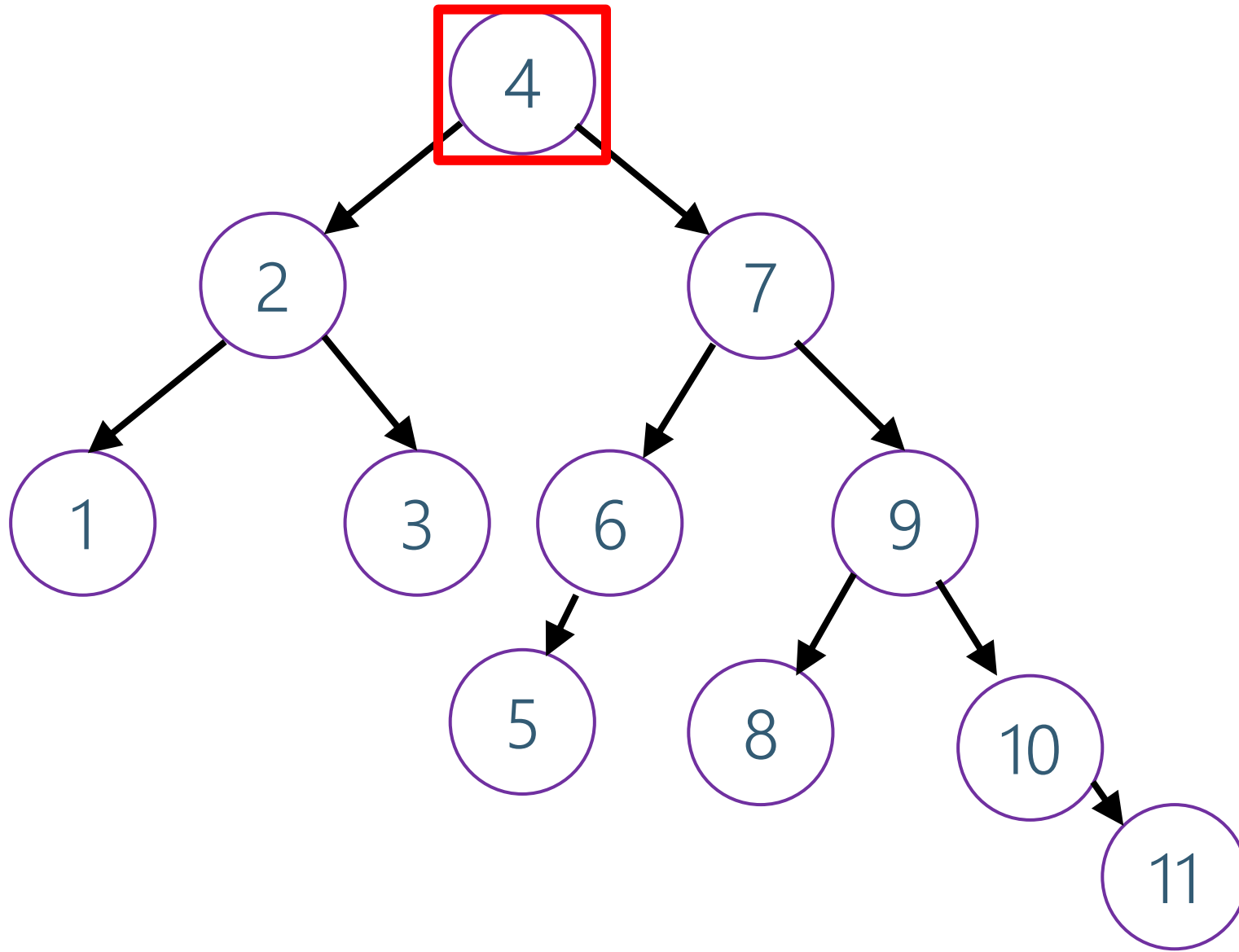
## Extra Example

---

# A Bigger AVL example: Insert 11

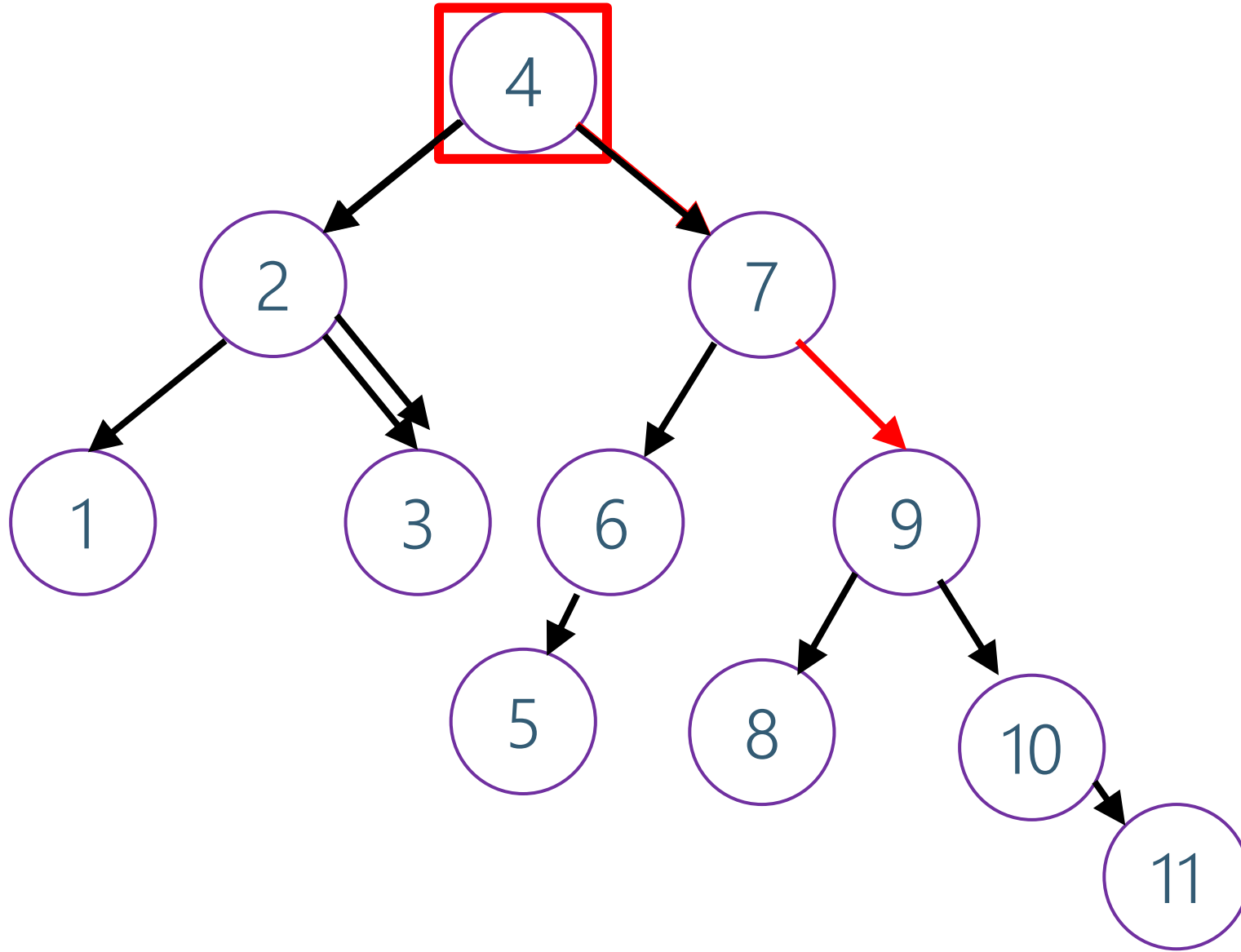


4 is imbalanced





Insert happened (to the right, to the right)  
Need a (single) left rotation



# A Bigger AVL example: Insert 11

